Homework 7: Q3

Name: Waiwai Kim, waiwaiki

1 Part (a): Proof Idea

Fix any topological ordering of G. Then consider the new graph G' that one obtains by removing from G all vertices that come before s in the chosen topological ordering. We are asked to argue that any $s-t$ path G is also an $s-t$ path in G'. The proof by contradiction is shown in the Week 10 Recitation notes.

2 Part (b): Algorithm Idea

We are asked to provide an algorithm that computes the shortest path in a Directed Acyclic Graph G with arbitrary weights. Note that weights can be positive or negative, which is different from Dijkstra's algorithm.

Computing the shortest path of DAG G can be done in two parts. First, build a topological ordering of G based on the algorithm provided in Sample Problem of HW5. Note that the provided algorithm returns a cycle when an input graph doesn't have a topological ordering. Since we are only concerned with DAGs which guarantee a topological ordering, we only need the first half of the Sample Problem's solution. Therefore, we will use TopologicalORdering(G) to produce a topological ordering.

Second, using the topological ordering, we will compute the distance from a starting node s to a node located to the right of s in a greedy manner. We'll set the distance of s to 0. Next, we will compute distances of the nodes that s has outgoing edges to. For example, if a node a has an incoming edge from s with a weight of 3, the distance from s to a would be $3 = 3 + 0$ and the incoming node is s. We could repeat this process from the starting node to the destination node. Because intermediate nodes can have incoming edges more than one, when we calculate a distance that is smaller than the distance previously computed, the algorithm needs to update the distance and the incoming node. The algorithm doesn't do anything if the newly computed distance is larger than the previously computed distance. When we arrive at the destination node, the algorithm has the shortest distance from s to t and the path by reverse traversing the incoming nodes from the destination node.

3 Part (b): Algorithm Details

First, let's discuss the input or the DAG given to the algorithm. Similar to the HW programming problems, the graph is expressed as a hash-table or a dictionary, where a key is a node and the value of a dictionary is a list of nodes that the node has outgoing edges to. Different from the HW programming problem, the list contains tuples, where the first element is the outbound node and the second element is the weight or the distance of the edge. For example, line 60 in Figure 1 shows that going from 1 to 2 costs 5 and going from 1 to 3 costs 3.

Next, let's take the TopologicalOrdering(G) shown in HW5 and slightly convert it specifically to the problem. The idea is to start a node that doesn't have any incoming edges and continue to add a node from a queue called NoIncoming to TopOrder. Line 10-12 iterates every edge in the graph and count then number of incoming edges for a node. Line 14-16 looks at IncomingEdges and add every node that doesn't have an incoming edge to the queue. Line 19 to 26 adds a node to TopOrder in the order that a node was added to the queue. It updates the number of incoming edges when an incoming node is added to the queue. At the end of the while loop, the algorithm returns a topological ordering. For a detailed explanation, please refer to the HW5.

Figure 1: Input Graph Dag - Q3 Part b

```
59
60      graph = {1: [(2,5), (3,3)],
61               2: [(3,2), (4,6)],
62               3:[(4,7),(5,4),(6,2)],
63               4:[(5,-1), (6,1)],
64               5:[(6,-2)],
65               6:[] }
```

Lastly, in Figure 3, the algorithm DagShort takes graph, topological ordering, starting node and finish node and computes the shortest path. In 32, a dictionary of nodes with values of distance is initialized at infinity. The infinity allows the algorithm to update the distance when a distance is computed for the first time. Line 33 sets the distance of the starting node to 0. In 35, we find out the starting index of s because we don't need to traverse any nodes to the left of the starting node. In 37, we will traverse each node starting from s to the end. In 38, we look at every node that the given node has an outbound edge to. node v has an outgoing edge to $vertex$ or $e[0]$ in line 39 and the cost of traveling from v to $vertex$ is $weight$ or $e[1]$. In line 42 and 43, if the current distance to $vertex$ is larger than the distance to the previous node plus $weight$, we will update the distance to $vertex$. The current distance to vertex is $dist[vertex][0]$ while the new distance is $dist[v][0] + weight$. If this is the case, we will update the incoming node to the previous node, which is shown in line 44. In line 47 to 53, the algorithm reverse traverses from the finish node to the starting node through the incoming node in order to build a path.

4 Part (b): Proof of Correctness Idea

First, we will use what we proved in part(a) that the shortest path $s-t$ of G still exists in $s-t$ in G' which doesn't have nodes that come before s. Based on this we know the solution we are looking for always exists to the right of s. Then, we can prove the correctness of the algorithm by contradiction by assuming that there is an optimal solution over a range of $s-t$ that our algorithm did not produce. Said differently, there is another path from s to t that is the optimal distance. At the end of the proof, we can show that any optimal solution is not better than the greedy solution of our algorithm, which proves that the greedy does in fact return an optimal solution.

5 Part (b): Proof Details

We know that the algorithm shown in Figure 2 is correct as it is shown as the solution to the sample problem in HW5. That leaves us proving that the algorithm shown in Figure 3 is correct. As we talked about previously, we will prove the correctness through contradiction. Let's $A = \{a_1, a_2, ..., a_k\}$ be the solution generated by our greedy algorithm and $O = \{o_1, o_2, ..., a_k\}$ be an optimal solution. This can be shown as the following: $\Sigma_i^k a_i > \Sigma_i^k o_i$. This entails that at least one edge of (o_i, o_{i+1}) in the optimal solution is shorter than and an edge (a_i, a_{i+1}) in our greedy algorithm. This means that there was e in our algorithm's line 38 of which weight is smaller than the weight of the edge that our greedy algorithm picked. However, this is a contraction as our algorithm is designed to pick the shortest edge for a given node v.

Figure 2: Produce Topological Order - Q3 Part b

```
4  def Toporder(G):
5
6      IncomingEdges = [0]*len(G)
7      NoIncoming = deque()
8      TopOrder = []
9
10     for v, e in G.items():
11         for edge in e:
12             IncomingEdges[edge[0]-1] += 1
13
14     for index in range(len(IncomingEdges)):
15         if IncomingEdges[index] == 0:
16             NoIncoming.append(index+1)
17
18
19     while NoIncoming :
20         current = NoIncoming.popleft()
21         TopOrder.append(current)
22
23         for e in G[current]:
24             IncomingEdges[e[0]-1] -=1
25             if IncomingEdges[e[0]-1] == 0 :
26                 NoIncoming.append(e[0])
27
28     return TopOrder
```

6 Part (b): Runtime Analysis

Please refer to the Sample Problem of HW5 for the runtime of the algorithm to produce a topological ordering for a DAG. Thus, we can safely claim that the runtime of the algorithm illustrated in Figure 2 is $O(m+n)$. Similarly we argue that the algorithm presented in Figure 3 is also $O(m+n)$. Line 37 of Figure 3 iterates every single node in the topological ordering produced by Figure 3. Line 38 of Figure 3 iterates every edge for a given node. Thus the asymptotic computational time of the algorithm is the same as iterating every single edge for a given graph represented in an adjacency list which is $O(m+n)$. The rest of the algorithm is insignificant as we are concerned with the asymptotic time and the runtime is smaller than $O(m+n)$. Thus we can argue that the runntime of algorithm is $O(m+n) = O(m+n) + O(m+n)$.

Figure 3: produce shortest path - Q3 Part b

```
30 def DagShort(G, top, s, f):
31
32     dist= {v: [float('infinity'), -1] for v,_ in G.items()}
33     dist[s][0] = 0
34
35     start_index = top.index(s)
36
37     for v in top[start_index:]:
38         for e in G[v]:
39             vertex = e[0]
40             weight = e[1]
41
42             if dist[vertex][0] > dist[v][0] + weight:
43                 dist[vertex][0] = dist[v][0] + weight
44                 dist[vertex][1] = v
45
46     #print(dist)
47     return_path = []
48     temp = f
49     return_path.insert(0, temp)
50
51     while temp != s:
52         temp = dist[temp][1]
53         return_path.insert(0,temp )
54
55     return return_path
```